割り算回路設計，あの手この手

新連載 1

回復法による基本的な除算回路の設計

鈴木昌治

デバイスの記事

関連データ

基本的な回路から性能（高速化）を考慮した回路まで，除算器の設計方法を数回にわたり解説する．除算器の設計は，とかく「めんどくさい」，「できれば避けて通りたい」と思われがちだが，ディジタル信号処理のアルゴリズムの複雑化などによって，その必要性は高まっている．第１回目の今回は，除算回路の基本的なアルゴリズムとして「回復法」を解説する．なお，本稿で設計した回路のソース・コードは本誌ホームページ（http://www.cqpub.co.jp/dwm/）からダウンロードできる．（編集部）

論理回路を設計していて除算回路が必要になったことがある，という方はそう少なくないと思います．純粋に数学的な演算もさることながら，ディジタル信号処理においても最近はアルゴリズムが複雑化しており，除算回路の必要性は高まっています（その一方で，除算を行わなくてもすむようなアルゴリズム上のくふうも進歩しているが…）．

除算回路の構成法は，古いものまで含めるとかなりのバリエーションがありますが，意外に知られていないのも事実です．除算回路は「手間のかかる回路」なので，ターゲット・デバイスの処理速度が遅かったころに少しでも速くしようと，かなりくふうが施された歴史があります．もちろん現在でも，進歩の途上にあります．多くのバリエーションをうまく使い分けることが，除算回路のおもしろみではないかと筆者は考えます．

●「除算器はかさばる」という先入観を捨ててみる

除算回路というと，「かさばるもの」と考えている方も少なくないようです．しかし，リアルタイム性が必要なく，のんびり演算することが許されるのであれば，分母の数のビット幅を持つ減算器とマルチプレクサ，それに簡単なシーケンサがあれば，分子がどんなに大きいけたの数であっても原理的には演算が可能です．条件によっては，決してかさばるものではありません．

筆者は画像の信号処理を扱うことが多いのですが，画像信号にはブランキング期間という画像情報のない期間があります．例えば，画像から特徴量（色や形状など，画像の特徴を数値化したもの）を抽出して信号処理の内容を変化させていくような適応型の処理が求められたケースを考えましょう．この場合，特徴量の処理をハードウェアで行うことが多いのですが，ここに除算が必要になるのは珍しいことではありません（自己相関の正規化などが良い例）．そのような場合，ブランキング期間にシーケンシャルな除算器をのんびり回す，というような構成手段が選択できれば，回路規模を増加させることなく機能を実現できます．

除算回路は「IP（intellectual property）コアとして提供されるもの」，あるいは「高級な論理合成ツールが吐き出してくれるもの」と割り切っている方もいるかと思います．しかし，除算回路のIPコアを選択する際に，構造として適当なものを選ぶには，除算についての最低限の知識が必要となります．まして，必要な性能の除算器が手に入らないのであれば，自分で何とかするしかありません．

食わず嫌いな方にあえて申し上げるなら，除算回路はそれほど難しいものではありません．わずかな基本的事項を理解するだけで，かなりの範囲に応用できるはずです．

本連載では，できるだけ多くの事例を挙げて除算回路のバリエーションを紹介していきたいと思います．同時に，その裏に隠されている数学的なからくりについても説明を加え，みなさんの応用に耐えうるものにしていこうと考えています．第1回の今回は，「回復法」を中心に除算の特性および基本回路を紹介しましょう．

1 まずは筆算のおさらい

筆算による除算というのは小学校で習った割り算の方法で，どなたも覚えがあると思います．**図1**に，その例を示しましょう．

手順は説明するまでもないのでしょうが，以下のとおりです．1回目に152の上位2けたである「15」と除数である「11」を比較し，11の1倍が引けそうなので商に1を立て，15－11＝4を演算します．2回目には余4を1けたシフト・アップ[注1]し，下位1けたを加えて「42」とします．11の3倍である33が引けそうなので商に3を立て，42－33＝9を演算します．2回の比較と減算によって除算が構成されていることになります．

●「除算」のとらえかたによって回路構成法も変わる

ここで，用語を以下のように統一します．本連載で各引き数を示す記号もこれに統一します．

- ***z***：**被除数（dividend）**
- ***d***：**除数（divisor）**
- ***q***：**商（quotient）**
- ***r***：**余（remainder）**

これらを用いて除算を表現すると，式(1)が成立します．

$$z=d\times q+r \quad \cdots\cdots(1)$$

そもそも除算とは何でしょうか．**図1**の例では，「152の中に11が何個とれるか」といった考えかたができます．分数的に考えれば，分母11と分子152の比と考えることもできます．そして，それぞれの考えかたに対応するように，除算回路の構成法が存在します．筆算による方法はもちろん前者で，その代表格になります．

式(1)を変形すると，式(2)になります．除算にこだわるなら，こちらの表現のほうが妥当かもしれません．

$$\frac{z-r}{d}=q \quad \cdots\cdots(2)$$

式(1)と式(2)は同じ関係を表現しているのですが，式(1)は乗算，式(2)は除算で表現されていることがわかります．つまり，乗算と除算は表裏一体の関係にあるわけです．冒頭で，除算回路は「手間のかかる回路」と言ったのですが，同じ関係にありながらなぜ除算は手間がかかるのでしょうか(「乗算は手間がかからない」という意味ではなく，あくまでも比較の問題)．

乗算は，“ sequential addition ”と言われます．部分積を求めて加算を繰り返せば答えに到達することができます．これに対して，除算は“ sequential subtraction ”と言われます．けた単位で減算を行い，そこでの余(部分余；partial remainder，以下prと表記)を求めるという操作を繰り返して真の商に近づけていきます．ここで，引き過ぎてはならない，という条件が必要になります．減算であることと引き過ぎてはならないという，二つの制約が除算をめんどうなものにしています(引き過ぎてもよいとする方法もあるが，その場合でも厳密な条件が必要になる．詳細は次回で説明)．

見かたを変えて式(1)を乗算と加算の複合として演算するとすれば，除数d，商q，余rが既知である必要があり，被除数zは未知数となります．しかし，一般に除算において既知であるのはd，zのみであり，q，rは未知数です．

引き数どうしは同じ関係にありながらアプローチが異なるため，演算の個性も変わってくるといえるのではないでしょうか．

2 回復法の定義と成立条件

「除算では減算を繰り返しながら商を求めるが，引き過ぎてはならない」と上述しました．この制約に厳密に従った方法が「回復法」という回路構成になります．「引けそうであれば引き，引けそうでなければ引かない」というルー

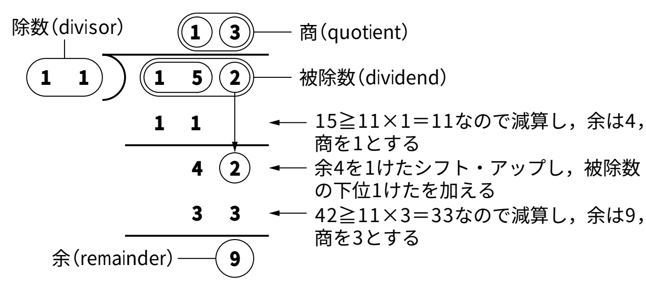


図1　筆算による除算

小学校のときに習った割り算のやりかた．演算回路を作るという視点から見ると，図の除算は2回の比較と減算によって構成されているということになる．

注1：シフト・アップとは，除数のけた位置を固定して，被除数(余も含めた)をけた上げしていくことを指す．図1では，被除数のけた位置を固定し，除数をシフト・ダウンしていることに注意．

ルの下で，減算とシフト・アップを繰り返します．

● **2進法で演算を行うと，“暗黙の乗算”が不要になる**

構造的には，「引けそうであるか否か」が比較にあたり，「引く」がそのまま減算にあたります．しかし，「引いてしまってから符号を調べて，負であれば引く前の値に戻す」という手順でも同じ結果が得られるので，ここから回復法の名があるわけです．構造としては，当然，コンパレータと減算器を兼用したものが一般的でしょう．

図1の例では10進法（10を法とする）で演算を行いましたが，論理回路でこれを実現するとなると，2進法（2を法とする）が基本になります．**図1**をそのまま2進法に変更して同一の演算を行うと，**図2**のようになります．

比較と減算の演算を1サイクルとして，5サイクル繰り返されていることがわかります．なお，図では比較の意味で，除数‘1011’を減算できないサイクルにおいても表示しています．

ここで，2を法として演算することの利点を明確にしておかなくてはなりません．2のべき乗や10など，2より大きな数を法として演算する場合には，逆の作用を及ぼすからです．

図1の2回目の演算で「42－33」を行っていますが，この33という数は除数11の3倍です．つまり，11×3を暗黙のうちに乗算しているわけです．これをそのまま回路化するとなると，当然乗算器が必要になります．一方，2を法として演算するとき，商として立つのは‘0’または‘1’です．つまり，減算に必要となる数は‘0’か除数そのものであり，乗算器は不要です．これが，2を法とする除算を行う場合の最大の利点と言えるでしょう．

さて，ここで**図3**に回復法のアルゴリズムを示します．まずは，**図2**に忠実に，比較と減算を分けて考えましょう．

ここで，先にも述べたとおり，比較と減算を兼用します．すると，フローチャートは**図4**のように書き換えられます．

● **「部分余＜2×除数」でなくては演算が成立しない**

回復法が除算アルゴリズムとして成立するには，条件があります．

図2で行った除算において，除数が11ではなく4だとどうなるのでしょう．けた位置をまったく変えずに行った場合の演算を**図5**に示します．正しくは，152÷4＝38（余0）なのですが，**図5**の演算では商が31で余が28となってしまいます．31×4＋28＝152で関係は正しいのですが，除算としては正しくありません．

こうなる原因は，**図3**における初期化にありました．**図3**のアルゴリズムは，$pr_i \geqq d$ の比較を行う段階で $pr_i < 2d$ で

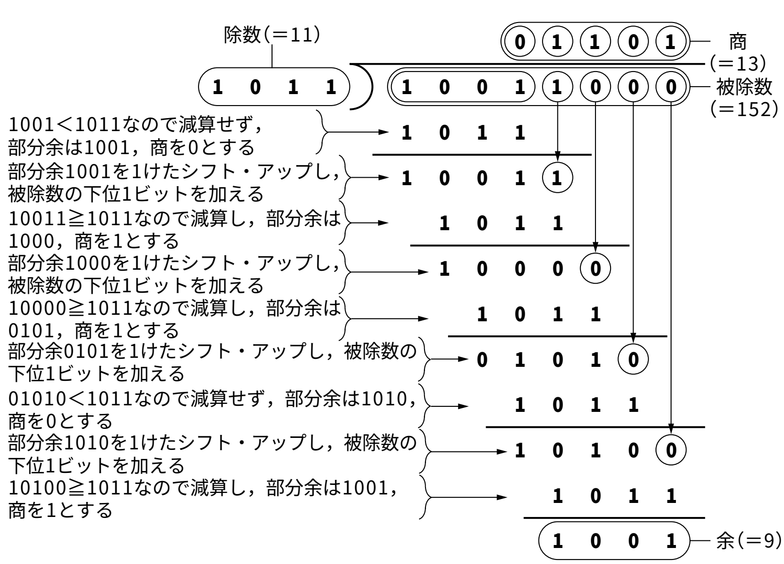


図2　図1の演算を2進法に変換
図1では10進法で割り算を行ったが，それをそのまま2進法に置き換えたものを図に示す．操作そのものは，図1となんら変わらない．

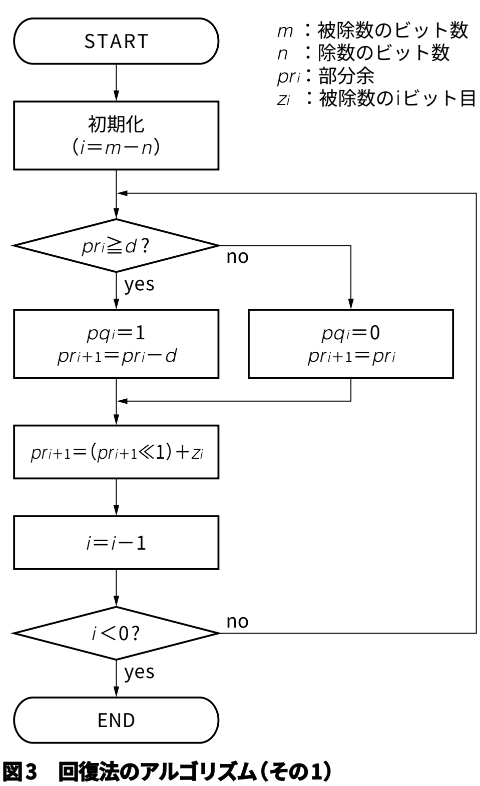


図3　回復法のアルゴリズム（その1）
図2に忠実に比較と減算を分けて考えた場合のアルゴリズム．

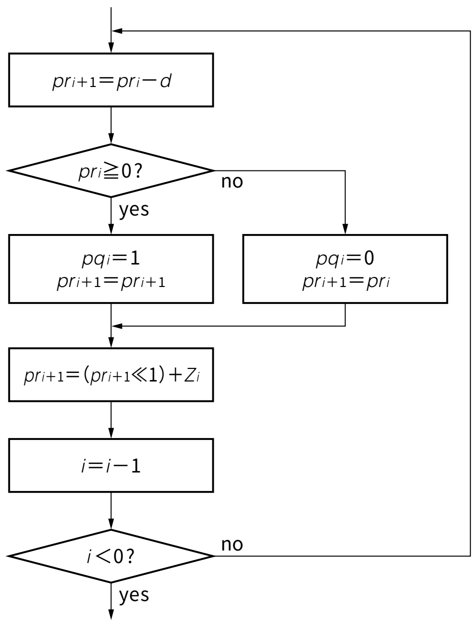


図4　回復法のアルゴリズム(その2)

「引いてしまってから符号を調べて，負であれば引く前の値に戻す」という手順で除算を行うアルゴリズム．比較と減算を兼用した形．繰り返しのループ部分のみ示す．

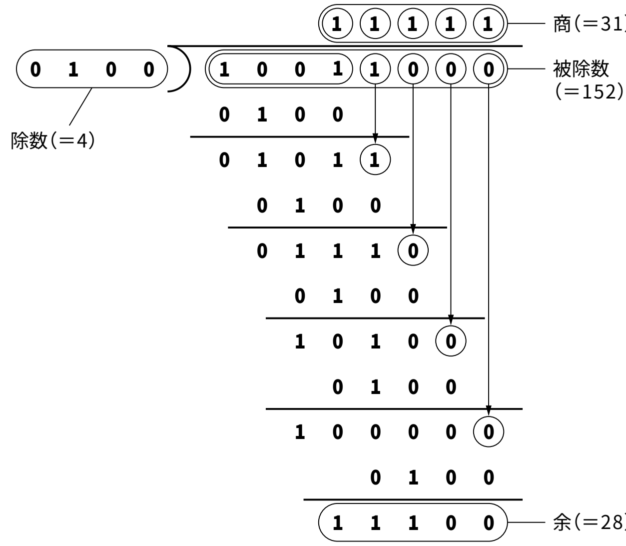


図5　回復法の失敗例

最初のサイクルで比較される9は除数4の2倍以上となっている．図3のアルゴリズムでは，部分余<2×除数という条件が成立しないと正しい結果が得られない．

なければならないのです．**図5**の例の場合，最初のサイクルで比較される9は除数4の2倍以上なので，ここでの商として2を立てたいところです．しかし，2を法としているかぎり，それはできません．

一般に，$pr_i \geqq 2d$の場合は$pr_{i+1} = pr_i - d \geqq d$が成立するため，次のステップでシフト・アップと加算を行うと，$pr_{i+1} \geqq 2d$となり悪循環は続いていきます．

逆に，$pr_i < 2d$を満たしておけば$pr_{i+1} = pr_i - d < d$が成立し，次のステップでも$p_{ri+1} < 2d$がかならず成立することになります．つまり，初期化の段階で$pr_{initial} < 2d$となるようにけたを合わせておけば，除算は破たんすることなく最後までまっとうされるわけです．

この条件を満たす方法は何通りかあります．被除数の上位に0を拡張し，かならず$pr_{initial} < 2d$を成立させるというのも一つの手です．**図5**の例では，被除数に0を3ビット分拡張すれば(“0001”となる)，$d > 0$の場合は条件を満たせます．この方法の利点は，めんどうな操作はいっさい必要ないことです．しかし，余分にiのループを回さなければなりません．

浮動小数点形式のように，上位側から見た最初の‘1’が現れるけたを左に詰めてしまうのも有効です．pr_iとdの両方にこの操作を施せば，商に0が立つ初期のむだなサイクルがほとんど排除されます．ただし，左詰めを行うためのハードウェアと，pr_iとdをシフトした分を最終的な商や余に対して補正する必要があります．いずれも一長一短があるので，目的に応じて使い分ければよいでしょう．

もう一つ，**図3**のアルゴリズムで除算を成立させる条件があります．それは，除数と被除数ともに正であることです．符号付きの演算も可能なのですが，このままのアルゴリズムでは不可能です．**図3**のアルゴリズムは，被除数から除数を減算し続けることで余を正側から0に近づけます．しかし，被除数が負であれば，除数を加算し続けることで負側から0に近づける操作になります．こうなると，条件に応じて加算と減算を使い分ける必要がでてくるわけで，アルゴリズムは複雑になります．ここでは，除算の基本を解説するのが目的なので，除数，被除数ともに正であると限定して話を進めましょう．

3　回復法をもとに回路を設計してみる

では，さっそく回復法による除算回路を設計してみましょう．とりあえず，被除数を16ビット，除数を8ビットの正の整数とします．$pr_{initial} < 2d$とするための方法としては，回路を簡略化するため，被除数の上位に0を拡張することにします．

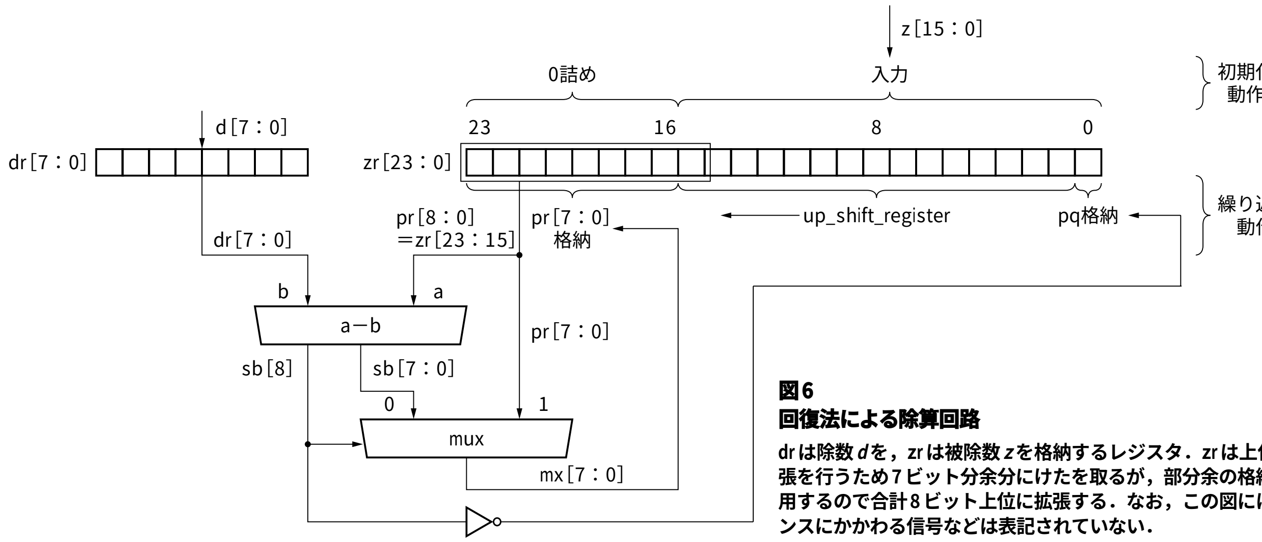


図6
回復法による除算回路
dr は除数 d を，zr は被除数 z を格納するレジスタ．zr は上位に 0 拡張を行うため 7 ビット分余分にけたを取るが，部分余の格納にも流用するので合計 8 ビット上位に拡張する．なお，この図にはシーケンスにかかわる信号などは表記されていない．

● リアルタイム性が低ければ基本的な回路構成で十分

図6にブロック図を示します．なお，この図にはシーケンスにかかわる信号などは表記されていないので注意してください．この回路の動作は，以下のようになります．

1) re（リセット・イネーブル）でシーケンスを初期化する．$i=0$ となり，start パルス待ちになる．
2) dr を除数 d の，zr を被除数 z の格納レジスタとする．zr は上位に 0 拡張を行うため 7 ビット分余分にけたを取るが，部分余の格納にも流用するので合計 8 ビット上位に拡張する．start パルスで z，d がそれぞれのレジスタに格納され，除算の初期化が終わる．z は，zr の下位側に格納される．また，$i=16$ にセットされる．
3) $pr-d$ が演算され，結果の符合によって商（部分商；partial quotient，以下 pq と表記）が決定され，このサイクルにおける部分余（pr）が選択される．次のサイクルに移行するにあたり，pq（1 ビット）は zr の LSB（least significant bit）に，pr（8 ビット）は zr の上位 8 ビットに戻される．さらに，zr[14：0] に格納されていた未使用の z は，1 ビットだけシフト・アップして，zr[14] は次の pr の LSB となる．ここで，$i=i-1$ が実行される．
4) 3) のサイクルを繰り返すことで，zr 上の z のデータは上位から 1 ビットずつ消化され，zr 上から消えていく．代わりに，zr の LSB に順次格納された pq が z のシフト・アップといっしょに上位へシフトされ，zr の下位側に蓄積される．
5) $i=0$ となったところで除算終了となる．zr[23：16] には余（r）が，zr[15：0] には商（q）が残り，z はすべてレジスタ上からなくなる．合計 17 サイクル（17 クロック）で終了となる（16 サイクルより 1 サイクル多いのは，z と d をレジスタにロードするサイクルが必要だから）．

この回路構成による Verilog HDL の RTL 記述を，**リスト1**に示します．

ここでは，減算と部分余を演算する系を 1 系統だけ用い，繰り返しによって除算を成立させました．リアルタイム性が必要なく，スループットが足りている場合であれば，この構成で十分です．

クロックの動作周波数が高い場合は，減算器の伝播遅延がネックになるでしょう．とくに，除数のビット幅が大きい場合，遅延が大きくなると同時に，符号ビットがドライブするマルチプレクサの制御のファンアウトも気になるところです．なにしろ，減算器のもっとも遅いパスである符号ビットでマルチプレクサを制御するわけですから…（このあたりの改善法は，次回で検討する）．

● スループットを上げたいときはパイプライン化

上述した方法は，除数が同じである限り，減算器とマルチプレクサの部分は変わらないので，被除数が大きくなっ

ても動作周波数は変化しません．回路規模は小さくなるのですが，サイクル数（スループット）が増えることになります．スループットを上げたい，あるいは毎クロック結果が欲しい場合には向きません．

そのような要求がなされたときのために，**図6**の構成にパイプライン化を施してみましょう．実はとても簡単に実現できるのです．

図7にその構成の概要を示します．**図6**の構成のままサブモジュール化し，zrにフィードバックしていた*pr*と*pq*を，次のサイクル用のサブモジュールにあるzrに渡すだけでよいのです．パイプライン化のため，*d*も次段に渡します．

この構造を16段繰り返すだけでパイプライン化は終了となります．必要に応じて出力段にレジスタを追加してください．*z*および*pq*をシフト・レジスタにてシフト・アップしていた部分は，モジュール間の接続によるシフトとなり，シーケンサも必要なくなります．これによってスループットを1とすることができます．

この回路構成によるVerilog HDLのRTL記述を，**リスト2**に示します．出力レジスタを付けなければ，レイテンシは16となります．

注2：例えば3を法とした場合，1けた当たりのとりうる値は{0，1，2}となる．これを表現するには2ビット必要であるが，2ビットで表現できる{3}は未使用のコードになる．このように，未使用の状態を持つコードをun_filled_codeと言う．4を法とすれば未使用コードはなくなり，filled_codeとなる．法には直接関係なく，例えば7ビットで0～99までしか状態が割り当てられていなければ，これもun_filled_codeである．

● 2より大きい数を法とする除算でレイテンシを抑える

図1と**図2**を比較すると明らかなのですが，10を法とした場合に2サイクルで終わった除算が，2を法とした場合には5サイクルかかっています．法として選択された数が大きくなると，除算を速く収束させることができるわけです．結果を得るまでのクロック数あるいはレイテンシを小さくしたい場合は効果的な方法です．

ここでは，2より大きい数を法とする除算の構成法を紹介しましょう．

2より大きい数を法とするとはいえ，3とか7を法として演算するのは論理回路の観点から適当ではありません．現状では，基本的に論理回路はバイナリ表現なので，4や8などの2のべき乗以外を法とすることはun_filled_code注2の生成につながり，効率が悪くなります．また，除算結果が3や7を法とした表現では周辺回路との整合が悪く，扱いもめんどうになります．ここでは，4を法として選択することにします．

図8(a)は，152÷11を4を法として演算しています．これはこれで正しい演算なのですが，1けたで表現する数は0，1，2，3となり，このままでは論理回路にできないことは明らかです．しょせんは2を法とするしかなく，1けた

リスト1　図6のVerilog HDL(div_restore_a.v)
演算中を示すbusyを出力してある．

```
module  div_restore_a    (
  q     , // quotient  [15:00]:integer
  r     , // remainder [07:00]:integer
  busy  , // busy

  z     , // dividend  [15:00]:integer
  d     , // divisor   [07:00]:integer
  start , // start (divider trigger)
  clk   , // clock
  re    );// reset ena

// io
  input [15:00]   z       ;
  input [07:00]   d       ;
  input           start   ;
  input           clk     ;
  input           re      ;

  output [15:00]  q       ;
  output [07:00]  r       ;
  output          busy    ;

// reg
  reg [04:00]     i       ;
  reg [23:00]     zr      ;
  reg [07:00]     dr      ;

// sequence(シーケンサ)
  wire  en_seq  = ( i > 0 ) ;

  always  @( posedge clk ) begin
        if      ( re )      i      <= 5'h00 ;
        else if ( start )   i      <= 5'h10 ;
        else if ( en_seq )  i      <= i - 1'b1 ;
  end

// function(演算処理部)
  wire  [08:00]  pr = zr[23:15] ;          // 前サイクルのシフト・アップ済み部分余
  wire  [08:00]  sb = pr - {1'b0,dr};      // 部分余ー除数
  wire           pq = ~sb[08] ;            // 減算結果の符号を反転し部分商とする
  wire  [07:00]  mx = ( sb[08] )? pr[07:00] :sb[07:00] ;
                                           // 減算前と後を減算結果により選択
  wire  [23:00]  zw = {mx,zr[14:00],pq} ;  // 次サイクル用zrレジスタ内容
                                           // シフト・アップ済み部分余
                                           // シフト・アップ済み被除数の未使用部
                                           // 部分商
  always  @( posedge clk ) begin
        if      ( start )   zr   <= {8'h00,z} ;
        else if ( en_seq )  zr   <= zw ;
  end

  always  @( posedge clk ) begin
        if      ( start ) dr  <= d;
  end

  assign  q    = zr[15:00] ;
  assign  r    = zr[23:16] ;
  assign  busy = en_seq ;

endmodule
```

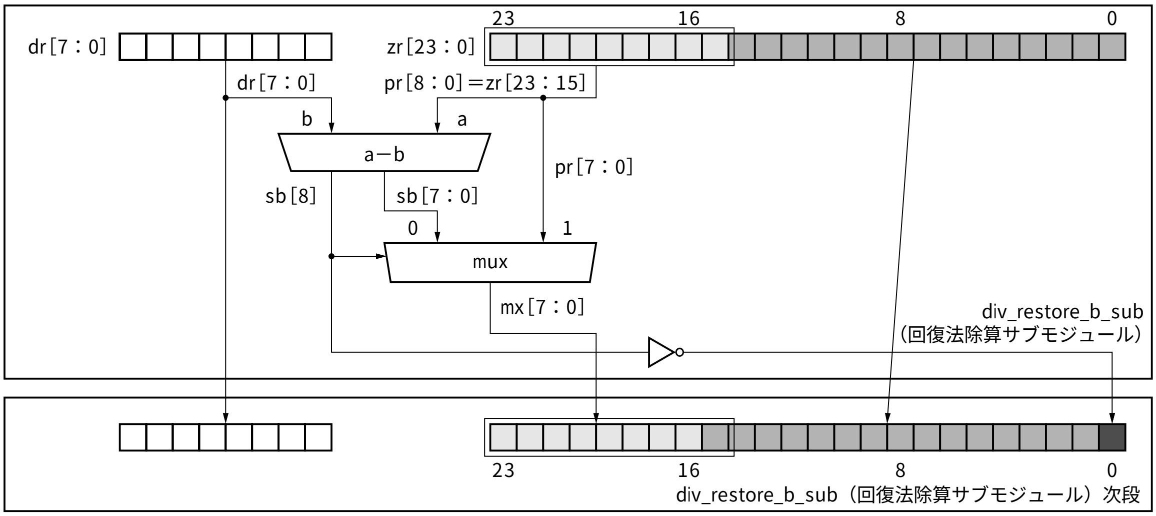


図7　回復法除算器のパイプライン化
図6の構成のままサブモジュール化し，zrにフィードバックしていたprとpqを，次のサイクル用のサブモジュールにあるzrに渡すだけでよい．

リスト2　パイプライン化を施した回路

```
module         div_restore_b      (
  q   ,  // quotient  [15:00]:integer
  r   ,  // remainder [07:00]:integer

  z   ,  // dividend [15:00]:integer
  d   ,  // divisor  [07:00]:integer
  clk); // clock

// io
  input [15:00]  z    ;
  input [07:00]  d    ;
  input          clk  ;

  output [15:00] q    ;
  output [07:00] r    ;

//wire
  wire [23:00]   zi_15  ,～中略～ ,zi_00 ;
  wire [07:00]   di_15  ,～中略～ ,di_00 ;
  wire [23:00]   zo_00 ;
  wire [07:00]   do_00 ;

// function
  assign zi_15  = {8'h00,z} ;
  assign di_15  = d ;

    div_restore_b_sub u15  (
        .zo  (zi_14 ),.do(di_14 ),
        .zi  (zi_15 ),.di(di_15 ),.clk(clk ));
    div_restore_b_sub u14  (
        .zo  (zi_13 ),.do(di_13 ),
        .zi  (zi_14 ),.di(di_14 ),.clk(clk ));
    div_restore_b_sub u13  (
        .zo  (zi_12 ),.do(di_12 ),
        .zi  (zi_13 ),.di(di_13 ),.clk(clk ));

    （中略）

    div_restore_b_sub u02  (
        .zo  (zi_01 ),.do(di_01 ),
        .zi  (zi_02 ),.di(di_02 ),.clk(clk ));
    div_restore_b_sub u01  (
        .zo  (zi_00 ),.do(di_00 ),
        .zi  (zi_01 ),.di(di_01 ),.clk(clk ));
    div_restore_b_sub u00  (
        .zo  (zo_00 ),.do(do_00 ),
        .zi  (zi_00 ),.di(di_00 ),.clk(clk ));

    assign q   = zo_00[15:00]  ;
    assign r   = zo_00[23:16]  ;

  endmodule
```

（**a**）メイン・モジュール（div_restore_b.v）

```
module   div_restore_b_sub       (
  zo   ,  // dividend out [23:00]:integer
  do   ,  // divisor out  [07:00]:integer

  zi   ,  // dividend in [23:00]:integer
  di   ,  // divisor in  [07:00]:integer
  clk);   // clock

// io
  input [23:00]    zi   ;
  input [07:00]    di   ;
  input            clk  ;

  output [23:00]   zo   ;
  output [07:00]   do   ;

// reg
  reg [23:00]      zr   ;
  reg [07:00]      dr   ;

// function（演算処理部）
  wire    [08:00]    pr    = zr[23:15] ;
  wire    [08:00]    sb    = pr - {1'b0,dr} ;
  wire               pq    = ~sb[08] ;
  wire    [07:00]    mx    = ( sb[08] )?  pr[07:00] :sb[07:00] ;
                     // 演算部分はdiv_restore_a.vとまったく同じ
  always  @( posedge clk ) begin
          dr   <= di  ;
          zr   <= zi  ;
  end

  assign  zo  = {mx,zr[14:00],pq} ; // 次のモジュールへの接続
  assign  do  = dr   ;

endmodule
```

（**b**）サブモジュール（div_restore_b_sub.v）

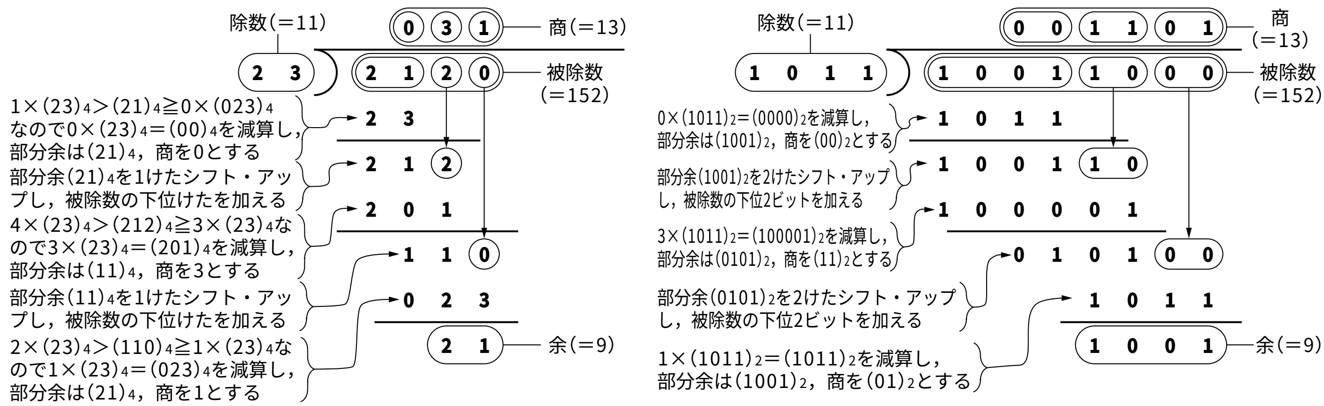


図8　4を法とする除算

(b)は，(a)を単純にバイナリ表現にしただけということがわかる．

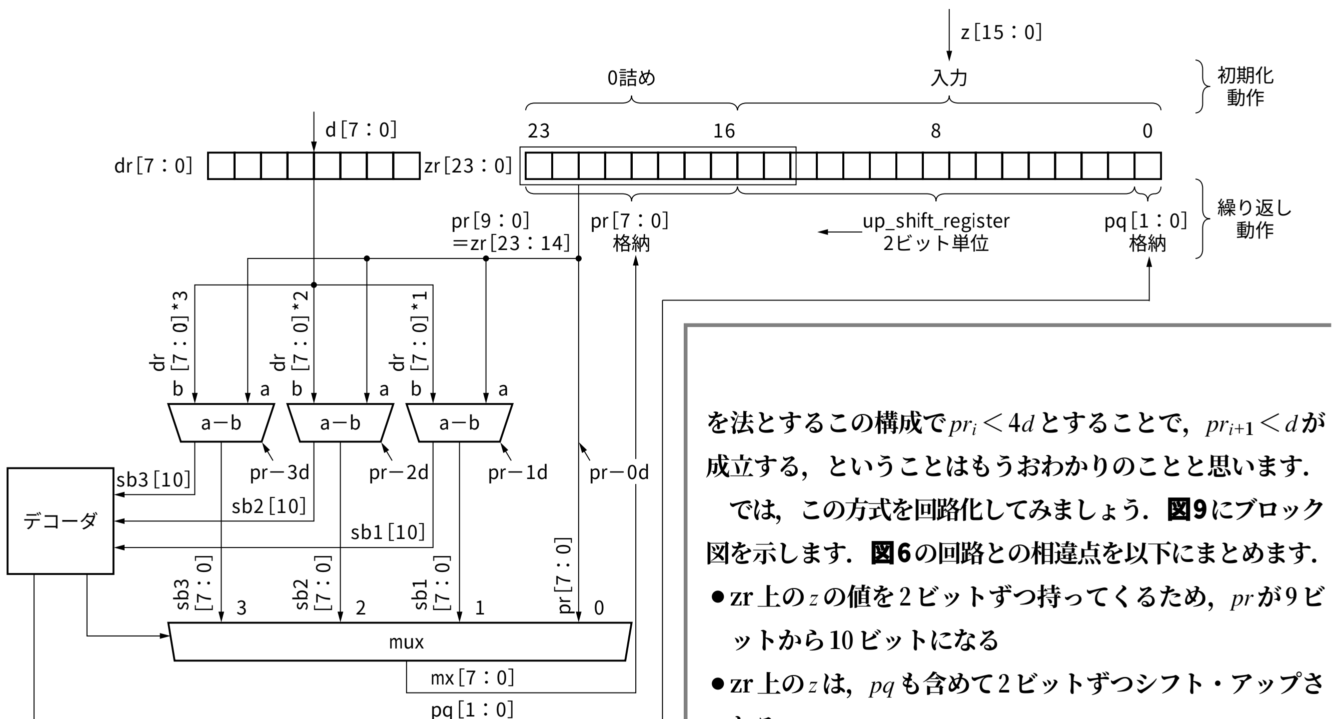


図9　4を法とする除算回路

図6の構成では17サイクルかかった除算が，9サイクルで実行できるようになる．ただし，減算器が3倍になり，マルチプレクサも大きいものが必要となる．

(1ビット)の表現する数は0，1となるからです．

4を法とした場合，1けたの表現する数は2を法とした場合の2けたで表現できます．要するに，2ビット単位で商を求める操作を行えばよいのです．**図8(b)**は，この考えに基づいてバイナリ表現の4を法とした除算を示します．これは，**図8(a)**を単純にバイナリ表現にしただけです．

先に説明した2を法とする除算の成立条件$pr_i < 2d$が，4を法とするこの構成で$pr_i < 4d$とすることで，$pr_{i+1} < d$が成立する，ということはもうおわかりのことと思います．

では，この方式を回路化してみましょう．**図9**にブロック図を示します．**図6**の回路との相違点を以下にまとめます．

- zr上のzの値を2ビットずつ持ってくるため，prが9ビットから10ビットになる
- zr上のzは，pqも含めて2ビットずつシフト・アップされる
- 減算器が1個から3個になる．それぞれd，$2d$，$3d$をprから減算するためのものである
- 減算結果の符号ビットが9ビット目から11ビット目に変更される．2を法としているときには減算が$pr_i < 2d$の条件の下で$pr_i - d$を行うのみだったので，減算器の出力が正のときはかならず$sb < d$が成立していた．しかし，4を法とするときは，例えば$3d \leqq pr_i < 4d$の場合に$pr_i - d$の減算器出力の10ビット目を符号ビットとしたのでは，値が正のときにも1が立っている可能性がある．符号ビットとして11ビット目を用意する必要がある
- 2 to 1のマルチプレクサから4 to 1のマルチプレクサに

リスト3　4を法とする除算回路（div_restore_c.v）

```
module  div_restore_c  (
  q     ,  // quotient  [15:00]:integer
  r     ,  // remainder [07:00]:integer
  busy  ,  // busy

  z     ,  // dividend [15:00]:integer
  d     ,  // divisor  [07:00]:integer
  start ,  // start (divider trigger)
  clk   ,  // clock
  re);     // reset ena

// io
  input [15:00]  z     ;
  input [07:00]  d     ;
  input          start ;
  input          clk   ;
  input          re    ;

  output [15:00] q     ;
  output [07:00] r     ;
  output         busy  ;

// reg
  reg [03:00]    i     ;
  reg [23:00]    zr    ;
  reg [07:00]    dr    ;

// sequence(シーケンサ)
  wire  en_seq  = ( i > 0 ) ;

  always  @( posedge clk ) begin
          if  ( re )          i    <= 4'h0 ;
          else if ( start )   i    <= 4'h8 ;
          else if ( en_seq )  i    <= i - 1'b1 ;
  end

// function(演算処理部)
  wire  [09:00]  pr    = zr[23:14] ;
// 前サイクルのシフト・アップ済み部分余
  wire  [09:00]  d1    =                dr       ;
  wire  [09:00]  d2    = ( dr << 1 )             ;
  wire  [09:00]  d3    = ( dr << 1 ) + dr        ;

  wire  [10:00]  sb1   = {1'b0,pr} - {1'b0,d1} ;
  wire  [10:00]  sb2   = {1'b0,pr} - {1'b0,d2} ;
  wire  [10:00]  sb3   = {1'b0,pr} - {1'b0,d3} ;

  function  [01:00] fpq ; // 部分商を求める関数
    input   sb1s  ,sb2s   ,sb3s   ;
    casex ({sb1s  ,sb2s  ,sb3s    })
          3'b1xx  :  fpq  = 2'h0  ;
          3'b01x  :  fpq  = 2'h1  ;
          3'b001  :  fpq  = 2'h2  ;
          3'b000  :  fpq  = 2'h3  ;
    endcase
  endfunction
  function [07:00]  fmx   ; // 部分余を求める関数
    input                sb1s  ,sb2s  ,sb3s  ;
    input  [07:00]  sb0   ,sb1    ,sb2   ,sb3   ;
    casex  ({sb1s  ,sb2s ,sb3s  })
            3'b1xx :    fmx    = sb0    ;
            3'b01x :    fmx    = sb1    ;
            3'b001 :    fmx    = sb2    ;
            3'b000 :    fmx    = sb3    ;
    endcase
  endfunction

  wire   [01:00]  pq  = fpq (sb1[10] ,sb2[10] ,sb3[10] );
  wire   [07:00]  mx  = fmx (sb1[10] ,sb2[10] ,sb3[10] ,
                          pr[07:00] ,sb1[07:00]
                            ,sb2[07:00] ,sb3[07:00] );
  wire   [23:00]  zw  = {mx,zr[13:00],pq} ;

  always  @( posedge clk ) begin
        if      ( start )  zr     <= {8'h00,z}  ;
        else if ( en_seq ) zr     <= zw   ;
      end

  always  @( posedge clk ) begin
        if      ( start )  dr     <= d  ;
  end

  assign  q     = zr[15:00] ;
  assign  r     = zr[23:16] ;
  assign  busy  = en_seq ;

endmodule
```

変更されている

● pqが1ビットから2ビットになる

この回路構成によるVerilog HDLのRTL記述を**リスト3**に示します．この回路構成によって，最初の回路構成で17サイクルかかった除算が9サイクルでできるようになります．パイプライン化も，前述したのとまったく同じ方法で実現できます．

結果を得るまでのサイクル数を減らすことはできましたが，減算器が3倍，マルチプレクサも大きいものになり，3dという今までには必要のなかった乗算が必要になりました．パイプライン化すれば，段数は半分になりますが，実質の回路規模は大きなものになります．8を法とする除算になると，この方式の延長にある限り，減算器は7倍になり，dの倍数を演算する乗算器もより多く必要となってきます（3d，5d，6d，7dは必要）．演算速度が上がるので，なんらかのリソースの投資は必要となるわけですが，できることなら回路規模は大きくしたくはないところです．

今回は，回復法を用いて除算回路の基本を押さえました．ここで紹介したバリエーションだけでも，かなりの要求にこたえることができるはずです．しかし，引き数が無符号なので，符号付きの除算を行いたい場合は除算前後で変換が必要というのが泣きどころですね．次回は，このあたりを追及してみましょう．

すずき・しょうじ
スタンダード・ロジック（株）
ssuzuki@stlc.co.jp

<筆者プロフィール>
諸般の事情で，実家がある新潟県でSOHOを始めました．生活のスタイルが根本から変わり，実に穏やかな時間を楽しんでいます．しごと場は鳥の声が聞こえる日本間で，設計の質にも良い影響が…？